Logistics

• MP2 due on Friday, April 9\textsuperscript{th}.

• MP3 will be released on Wednesday, April 14\textsuperscript{th}

• HW5 is due on Thursday, April 15\textsuperscript{th}.
  • You should be able to answer Q1, Q2, Q3a and Q3b right away.
  • Q3 (parts c, d, e) have been slightly updated.
  • You should be able to answer Q3 entirely after today’s class.
Agenda for today

• Transaction Processing and Concurrency Control
  • Chapter 16
    • Transaction semantics: ACID
    • Isolation and serial equivalence
    • Conflicting operations
    • Two-phase locking
    • Deadlocks
    • Timestamped ordering

• First focus on transactions executed on a single server.
• Look into distributed transactions later (Chapter 17)
Transaction Properties: ACID

- **Atomic**: all-or-nothing
  - Transaction either executes completely or not at all
- **Consistent**: rules maintained
- **Isolation**: multiple transactions do not interfere with each other
  - Equivalent to running transactions in isolation
- **Durability**: values preserved even after crashes
Isolation

How to prevent transactions from affecting each other?

• Option 1: Execute them serially at the server (one at a time).
  • e.g. through a global lock.
  • But this reduces number of concurrent transactions

• Instead of targeting serial execution, target serial equivalence.
  • Conflicting operations executed in the same transaction order.
  • How do we ensure this?

• Option 2: at commit point, check if serial equivalence violated. If yes, abort transaction.
  • Too many aborts. Lower transaction throughput.

Goal: increase concurrency and transaction throughput while maintaining correctness (ACID).
Concurrency Control: Two approaches

- **Pessimistic**: assume the worst, prevent transactions from accessing the same object
  - E.g., Locking

- **Optimistic**: assume the best, allow transactions to write, but check later
  - E.g., Check at commit time
Concurrency Control: Two approaches

• **Pessimistic**: assume the worst, prevent transactions from accessing the same object
  • E.g., Locking

• **Optimistic**: assume the best, allow transactions to write, but check later
  • E.g., Check at commit time
Pessimistic: Locking

- Grabbing a global lock is wasteful
  - what if no two transactions access the same object?

- Each object has a lock
  - can further improve concurrency.
  - reads on the same object are non-conflicting.

- Per-object read-write locks.
  - Read mode: multiple transactions allowed in
  - Write mode: exclusive lock
When to release locks?

- We can have per-object locks in two modes to increase concurrency.
- Grab the object's lock in the appropriate mode when trying to access an object.
- When to release locks?

<table>
<thead>
<tr>
<th>lock_mode</th>
<th>lock_operation</th>
<th>unlock_operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>write_lock(A)</td>
<td>write(A)</td>
<td>unlock(A)</td>
</tr>
<tr>
<td>read_lock(A)</td>
<td>read(A)</td>
<td>unlock(A)</td>
</tr>
</tbody>
</table>

Is this a good idea?
When to release locks?

- We can have per-object locks in two modes to increase concurrency.
- Grab the object's lock in the appropriate mode when trying to access an object.
- **When to release locks?**
Guaranteeing Serial Equivalence with Locks

- **Two-phase locking**
  - A transaction cannot acquire (or promote) any locks after it has started releasing locks
  - Transaction has two phases
    1. **Growing phase**: only acquires or promotes locks
    2. **Shrinking phase**: only releases locks
      - **Strict two phase locking**: releases locks only at commit point
Two-phase Locking

write_lock(A)
write(A)
unlock(A)

read_lock(A)
read(A)
unlock(A)

Not allowed with two-phase locking

Not serially equivalent
Two-phase Locking

write_lock(A)
write(A)
unlock(A)

read_lock(A)
read(A)
unlock(A)

write_lock(A) blocked

write(A)
unlock(A)

Serially equivalent!
Why two-phase locking => Serial Equivalence?

- Proof by contradiction
- Assume two phase locking system where serial equivalence is violated for some two transactions T1, T2

Two facts must then be true:
- (A) For some object O1, there were conflicting operations in T1 and T2 such that the time ordering pair is (T1, T2)
  - (A) => T1 released O1’s lock and T2 acquired it after that
    => T1’s shrinking phase is before or overlaps with T2’s growing phase
  
- (B) For some object O2, the conflicting operation pair is (T2, T1)
  - (B) => T2’s shrinking phase is before or overlaps with T1’s growing phase

But both these cannot be true!
Lost Update Example with 2P Locking

Transaction T1
read_lock(x)
\[ x = \text{getSeats}(ABC123); \]

if\( (x > 1) \)
\[ x = x - 1; \]
write_lock(x)  \textit{Blocked!}
write(x, ABC123);

commit

Transaction T2
read_lock(x)
\[ x = \text{getSeats}(ABC123); \]

if\( (x > 1) \)
\[ x = x - 1; \]
write_lock(x)  \textit{Blocked!}
write(x, ABC123);

commit

Deadlock!
Downside of Locking

• Deadlock!
## Deadlock Example

### Transaction T1
- `read_lock(x)`
- `x = getSeats(ABC123);`
- `if(x > 1)`
  - `x = x - 1;`
- `write_lock(x) Blocked!`
- `write(x, ABC123);`
- `unlock(x)`
- `commit`

### Transaction T2
- `read_lock(x)`
- `x = getSeats(ABC123);`
- `if(x > 1)`
  - `x = x - 1;`
- `write_lock(x) Blocked!`
- `write(x, ABC123);`
- `unlock(x)`
- `commit`

---

**Deadlock!**

Diagram:
- T1
- T2
  - Wait for
  - Wait for
When do deadlocks occur?

• 3 necessary conditions for a deadlock to occur
  1. Some objects are accessed in exclusive lock modes
  2. Transactions holding locks are not preempted
  3. There is a circular wait (cycle) in the Wait-for graph

• “Necessary” = if there’s a deadlock, these conditions are all definitely true

• (Conditions not sufficient: if they’re present, it doesn’t imply a deadlock is present.)
Combating Deadlocks

1. Lock all objects in the beginning in a single atomic step.
   • no circular wait-for graph created (3rd deadlock condition breaks)
     - may not know of all operations a priori.

2. Lock **timeout**: abort transaction if lock cannot be acquired within timeout
   • (2nd deadlock condition breaks)
     - Expensive; leads to wasted work
     - How to determine the timeout value?
       • Too large: long delays
       • Too small: false positives.

3. Deadlock **Detection**:
   • keep track of Wait-for graph, and find cycles in it (e.g., periodically)
   • If find cycle, there’s a deadlock
     => Abort one or more transactions to break cycle (2nd deadlock condition breaks)
Concurrency Control: Two approaches

- Pessimistic: assume the worst, prevent transactions from accessing the same object
  - E.g., Locking

- Optimistic: assume the best, allow transactions to write, but check later
  - E.g., Check at commit time
Optimistic Concurrency Control

- Increases concurrency more than pessimistic concurrency control
- Used in Dropbox, Google apps, Wikipedia, key-value stores like Cassandra, Riak, and Amazon’s Dynamo
- Preferable than pessimistic when conflicts are expected to be rare
  - But still need to ensure conflicts are caught!
First cut approach

- Most basic approach
  - Write and read objects at will
  - Check for serial equivalence at commit time
  - If abort, roll back updates made
  - An abort may result in other transactions that read dirty data, also being aborted
    - Any transactions that read from those transactions also now need to be aborted
      😞 Cascading aborts
Timestamped ordering

- Assign each transaction an id
- Transaction id determines its position in serialization order.
- Ensure that for a transaction T, both are true:
  1. T’s write to object O allowed only if transactions that have read or written O had lower ids than T.
  2. T’s read to object O is allowed only if O was last written by a transaction with a lower id than T.
- Implemented by maintaining read and write timestamps for the object
- If rule violated, abort!
- Never results in a deadlock! Older transaction never waits on newer ones.
Timestamped ordering: per-object state

- Committed value.
- Transaction id (timestamp) that wrote the committed value.
- Read timestamps (RTS): List of transaction ids (timestamps) that have read the committed value.
- Tentative writes (TW): List of tentative writes sorted by the corresponding transaction ids (timestamps).
  - Timestamped versions of the object.
## Timestamped ordering rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>$T_c$</th>
<th>$T_i$</th>
</tr>
</thead>
</table>
| 1.   | write | read  | $T_c$ must not write an object that has been read by any $T_i$ where $T_i > T_c$  
This requires that $T_c \geq$ the maximum read timestamp of the object. |
| 2.   | write | write | $T_c$ must not write an object that has been written by any $T_i$ where $T_i > T_c$  
This requires that $T_c >$ write timestamp of the committed object. |
| 3.   | read  | write | $T_c$ must not read an object that has been written by any $T_i$ where $T_i > T_c$  
This requires that $T_c >$ write timestamp of the committed object. |
Timestamped ordering: write rule

Transaction $T_c$ requests a write operation on object $D$
if ($T_c \geq$ max. read timestamp on $D$
  && $T_c >$ write timestamp on committed version of $D$)
   \textit{Perform a tentative write on $D$:}
      \begin{itemize}
      \item If $T_c$ already has an entry in the TW list for $D$, update it.
      \item Else, add $T_c$ and its write value to the TW list.
      \end{itemize}
else
   
   \textit{abort transaction $T_c$}
   \textit{//too late; a transaction with later timestamp has already read or written the object.}
Timestamped ordering: write rule

(a) \( T_3 \) write

Before
\[ T_2 \]

After
\[ T_2 \quad T_3 \]

Time

(b) \( T_3 \) write

Before
\[ T_1 \quad T_2 \]

After
\[ T_1 \quad T_2 \quad T_3 \]

Time

Key:
\[ \text{T_i} \]
Committed
Tentative

T_1 < T_2 < T_3 < T_4

Read timestamps not shown in this example.
(assume zero reads)

(c) \( T_3 \) write

Before
\[ T_1 \quad T_4 \]

After
\[ T_1 \quad T_3 \quad T_4 \]

Time

(d) \( T_3 \) write

Before
\[ T_4 \]

After
\[ T_4 \]

Transaction aborts

Time
Transaction $T_c$ requests a read operation on object $D$

if ($T_c > \text{write timestamp on committed version of } D$) {
    $D_s = \text{version of } D \text{ with the maximum write timestamp that is } \leq T_c$
    \text{//search across the committed timestamp and the TW list for object } D.
    if ($D_s$ is committed)
        read $D_s$ and add $T_c$ to RTS list (if not already added)
    else
        if $D_s$ was written by $T_c$, simply read $D_s$
        else
            wait until the transaction that wrote $D_s$ is committed or aborted, and
            reapply the read rule.
            \text{// if the transaction is committed, } T_c \text{ will read its value after the wait.}
            \text{// if the transaction is aborted, } T_c \text{ will read the value from an older transaction.}
} else
    abort transaction $T_c$
    \text{//too late; a transaction with later timestamp has already written the object.}
Timestamped ordering: read rule

(a) T₃ read

Selected

T₂

read proceeds

Time

(c) T₃ read

Selected

T₁

T₂

read waits

Time

(b) T₃ read

Selected

T₂

T₄

read proceeds

Time

(d) T₃ read

T₄

Transaction aborts

Time

Key:

Tᵢ

Committed

Tᵢ

Tentative

T₁ < T₂ < T₃ < T₄
Suppose $T_4$ is ready to commit. Must wait until $T_3$ commits or aborts.

When a transaction is committed, the committed value of the object and associated timestamp are updated, and the corresponding write is removed from TW list.
Lost Update Example with Timestamped Ordering

Transaction T1

\[ x = \text{getSeats}(\text{ABC123}); \]
\[ \text{if}(x > 1) \]
\[ x = x - 1; \]
\[ \text{write}(x, \text{ABC123}); \]
\[ \text{commit} \]

Transaction T2

\[ x = \text{getSeats}(\text{ABC123}); \]
\[ \text{if}(x > 1) \]
\[ x = x - 1; \]
\[ \text{write}(x, \text{ABC123}); \]
\[ \text{commit} \]

ABC123: state
committed value = 10
committed timestamp = 0

RTS:
TW:
### Next Example with Timestamped Ordering

<table>
<thead>
<tr>
<th>Transaction T1</th>
<th>Transaction T2</th>
<th>ABC123: state committed value = 10 committed timestamp = 0 RTS: TW:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = \text{getSeats}(\text{ABC123}); )</td>
<td>( x = \text{getSeats}(\text{ABC123}); )</td>
<td>ABC789: state committed value = 5 committed timestamp = 0 RTS: TW:</td>
</tr>
<tr>
<td>( y = \text{getSeats}(\text{ABC789}); )</td>
<td>( y = \text{getSeats}(\text{ABC789}); )</td>
<td></td>
</tr>
<tr>
<td>write((x-5, \text{ABC123});)</td>
<td>print(\text{“Total:” } x+y); )</td>
<td></td>
</tr>
<tr>
<td>write((y+5, \text{ABC789}); )</td>
<td>commit</td>
<td></td>
</tr>
<tr>
<td>commit</td>
<td>commit</td>
<td></td>
</tr>
</tbody>
</table>
Concurrency Control: Summary

• How to prevent transactions from affecting one another?
• Goal: increase concurrency and transaction throughput while maintaining correctness (ACID).
• Target serial equivalence.
• Two approaches:
  • Pessimistic concurrency control: locking based.
    • read-write locks with two-phase locking and deadlock detection.
  • Optimistic concurrency control: abort if too late.
    • timestamped ordering.
Next Class

• Distributed Transactions.